(2)

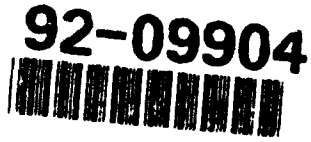
Yale University
Department of Computer Science

Crystal Reference Manual
Version 3.0

Michel Jacquemin J. Allan Yang

YALEU/DCS/TR-840
March 1991

This work has been supported in part by the Office of Naval Research under Contract No. N00014-90-J-1987 and the National Science Foundation under Contract No. 622A-31-47097.



Crystal Reference Manual

Version 3.0

Michel Jacquemin J. Allan Yang

Department of Computer Science
Yale University
New Haven, CT 06520-2158
jacquemin@cs.yale.edu, yang@cs.yale.edu

January, 1991

Foreword

This manual describes the new syntax of Crystal, a strongly-typed lexically-scoped functional language for programming massively parallel machines, and an implementation of a *Crystal interpreter* based on T, a dialect of Scheme. It is a reference manual, intended to define the language for which the interpreter is implemented. It is not intended as an introduction or a tutorial. The reader is expected to have some basic knowledge about Crystal [1, 2].

1 Language Definition

In this section, we describe the syntactic structures and their associated meanings of Crystal. Examples are given for illustration.

1.1 Notational Convention

We follow the conventions below in describing the syntax of Crystal.

$[a]$	a is optional
$\{a\}$	zero or more repetition of a
$a \mid b$	choice between either a or b
a	actual text in the program
(a)	grouping

This work has been supported in part by the Office of Naval Research under Contract No. N00014-90-J-1987 and the National Science Foundation under Contract No. 622A-31-47097.

1.2 Programs and Definitions

A Crystal *program* contains a set of possibly mutually recursive *definitions* and optional output expressions, having the syntax:

program → (*definition* [*doc*] | ?*expression*) {*definition* [*doc*] | ?*expression*}
definition → *identifier* [: *type-exp*] = *expression*
doc → doc{ anything except a right curly brace }

The optional *type-exp* in the *definition* specifies the type of the identifier. Each ?*expression* is an output expression whose value will be printed out.

Example Below is an example of a simple Crystal program:

```
n = 10
D = interval(1,n)
F = df (x):D {2 * x}
?F
?D
```

1.3 Expressions

Expressions are the most basic construct in Crystal programs. It is used to express all the semantic objects of Crystal. The *basic semantic objects* of Crystal consist of integers, floating point numbers, booleans, and strings. The *composite semantic objects* of Crystal consists of tuples, tagged values, index domains, data fields, lists, sets, and functions over these objects. Expressions have the following syntax:

expression → *constant* | *identifier* | *tuple* | *tagged-exp* | *expression* : *type-exp*
| *unary-op expression* | *expression binary-op expression*
| *conditional* | *function-abst* | *function-appl*
| *domain-exp* | *dfield-exp* | *list-exp* | *set-exp*
| (*expression*) | *expression* **where**{{*definition*}}

The nonterminals *constant* and *identifier* will be defined in Section 1.11. The *tuple* and *tagged-exp* are tuples and tagged values, which will be defined in Section 1.5. The expression *expression* : *type-exp* is a typed expression in which a type is explicitly specified for the expression; it will be defined in Section 1.10. The *unary-op* is the family of unary prefix operators and the *binary-op* is the family of binary infix operator; they will be defined in Section 1.11. The *conditional* is the conditional expression, which will be defined in Section 1.4. The *function-abst* and *function-appl* are function abstraction and application, which will be defined in Section 1.6. The *domain-exp* is for index domain expressions, which will be defined in Section 1.7. The *dfield-exp* is for data field expressions, which will be defined in Section 1.8. The *list-exp* is for list expressions and the *set-exp* is for set expressions; they will be defined in Section 1.9. The (...) is for grouping. The **where**{...} construct is for introducing local definitions for identifiers occurring in the preceding expression.

1.4 Conditional

The conditional expression has the form:

if *expression* **then** *expression*
{**||** *expression* **then** *expression*}
[[**||**] **else** *expression*]
fi

The expressions before **then** are called *guards* and those after **then** are called *consequences*. Guards should denote boolean values. The expression after **else** is called the *default* expression, which is optional. The value of a conditional expression is the first consequence with a true guard. When there is no true guard, the value of the conditional is the default expression if there is one, otherwise it is undefined.

Example Below is a conditional expression that picks out the maximum among **x**, **y**, and **z**:

```
if x > y and x > z then x
|| y > z then y
|| else z
fi
```

1.5 Tuples and Tagged Values

Tuples are used for expressing elements of product types or product domains (explained later). Tagged values are used for expressing elements of summed types or summed domains (explained later). They have the following syntax:

tuple → ([*expression* , *expression* {, *expression*}])
tagged-exp → *tag*^*tuple*
tag → *expression*

There is no single-element tuple. They are considered to be equivalent as the element itself. In the expression *tag*^*tuple*, the *tag* should denote a natural number, and the length of *tuple* (defined below) should be the same as the number of constituent domains of the summed domain.

Let t and u be the values denoted by *tag* and *tuple*, respectively, and the product domain $d_0 \times \ldots \times d_{n-1}$ be the type of *tuple*. The meaning of *tag*^*tuple* is $\iota(u(t))$, where $u(t)$ is the t-th element of the tuple u and ι is the injection function from the domain $d_t, 0 \leq t < n$, to the summed domain $d_0 + \ldots + d_{n-1}$.

The syntax for the tagged expression is somewhat unconventional, but it provides more information for the analysis performed by the parallelizing compiler. It also makes expressions that depend on the value of the tag more compact, as is shown in the defintion for **f3** in the example in section 1.6.

Selectors and Operators for Tuples and Tagged Expressions

Let $e, e_0, e_1, e_2, \ldots$ be expressions. The following selectors and operators are defined for tuples and tagged expressions:

$(e_0, e_1, \ldots, e_{n-1})(i)$ denotes the value of e_i, where $0 \leq i < n$.

length$(e_0, e_1, \ldots, e_{n-1})$ denotes n.

tag_of$(e\hat{}(e_0, e_1, \ldots, e_{n-1}))$ denotes the value of e.

value_of$(e\hat{}(e_0, e_1, \ldots, e_{n-1}))$ denotes the value of $(e_0, e_1, \ldots, e_{n-1})$.

Example Below are examples of tuples and tagged values:

```
tup = (1,2,3)
tv = 1^tup
```

We will have the following: `tup(0)` is 1, `length(tup)` is 3, `tag_of(tv)` is 1, and `value_of(tv)` is (1,2,3).

1.6 Function Abstraction and Application

Function abstraction and application have the following syntax:

function-abst → **fn** *formal* : *type-exp* [| *filter*] { *expression* }
function-appl → *expression* (*expression*) | *expression tuple*
formal → *identifier* | ([*identifier* {, *identifier*}]) | *formal* ^ *formal*
filter → *expression*

Function abstraction is used for expressing functions, which are first class objects. A filter is an expression denoting a boolean value. When there is a filter in the function abstraction, the actual argument is checked against the filter. If it does not pass the filter then the returned value is undefined. The formal can be of the form of a single identifier, a tuple of identifiers, or identifiers in a tagged expression form. The actual argument will be destructured properly according to the form of the formal.

Example Below are examples of functions:

```
f1 = fn x:nat { x*2 }
f2 = fn (x,y) :  prod_type(nat,nat) { x + y }
f3 = fn (t^(u,v)) :  sum_type(bool,nat) { t^(not u, v * 3) }
fac = fn n :  int | n >= 0 {
              if n = 0 then 1
              || else n * fac(n-1)
              fi }
```

We will have the following: `f1(3)` is 6, `f2(2,3)` is 5, `f3(0^(true,nil))` is ι_1(false), where ι_1 is the injection function from `bool` to `sum_type(bool,nat)`; and `f3(1^(nil,3))` is $\iota_2(9)$, where ι_2 is the injection function from `nat` to `sum_type(bool,nat)`. The constant `nil` is used to express an undefined value. The function `fac` is the usual factorial function, in which a filter is used to make sure the argument is non-negative.

1.7 Index Domains

An *index domain* is a set of points indexed in some way. Index domain expressions have the following syntax:

domain-exp → **interval**(*expression*,*expression*)
| **dom**{ *expression* {; *expression*} }
| **prod_dom**(*domain-exp*, *domain-exp* {, *domain-exp*})
| **sum_dom**(*domain-exp*, *domain-exp* {, *domain-exp*})
| **gen_prod_dom** *formal* : *domain-exp* [| *filter*] { *domain-exp* }
| **gen_sum_dom** *formal* : *domain-exp* [| *filter*] { *domain-exp* }
| **dom**[*formal* : *domain-exp* | *filter*]

Let e_1, e_2, ...,e_n be expressions denoting integer values. An *interval* index domain is constructed by `interval(`e_1`,`e_2`)`, with e_1 being its *lower bound* and e_2 being its *upper bound*. It contains $e_2 - e_1 + 1$ points indexed from e_1 to e_2. If $e_2 < e_1$, then `interval(`e_1, e_2`)` is empty. An *enumerated* index domain is constructed by `dom{`e_1`;`e_2`;...;`e_n`}`. It contains n points indexed by the values of e_1,..., e_n. Having duplicate elements is permitted and data fields defined over them will have only one value for the same index. Note that there should be *no* space between `dom` and `{`.

Let d_0, ...,d_n be index domain expressions. The product domain and sum domain of d_0,..., d_n are denoted by the expressions `prod_dom(`d_0`,...,`d_n`)` and `sum_dom(`d_0`,...,`d_n`)`, respectively. Let x_0 be an expression denoting an element in d_0, and x_n an element of d_n, the tuple `(`x_0`,...,`x_n`)` is an element of `prod_dom(`d_0`,...,`d_n`)`. The tagged expressions `0^(`x_0`,nil,...,nil)`,..., `n^(nil,...,nil,`x_n`)` are elements of `sum_dom(`d_0`,...,`d_n`)`.

Let x be a formal, d be a domain expression, e be another domain expression with occurrences of x, and f be a filter (i.e., a boolean expression) with occurrences of x. The `gen_prod_dom` and `gen_sum_dom` are for constructing general product and sum domains. The expression `gen_prod_dom` x`:`d{e} denotes the product domain of all domains produced by instantiating the formal x in e with every point in the domain d. When there is a filter expression f, x is only instantiated with points in d that pass f (i.e. f evaluates to true). Similarly for `gen_sum_dom`.

Let x, d, f be as described above. The expression `dom[`x`:`d`|`f`]` is a *domain restriction* denoting the restricted domain that contains only points of d that pass the filter. Note that there should be *no* space between `dom` and `[`.

Operators on Index Domains

Let d be a domain expression, and e be an expression denoting an domain element (i.e., an integer, a tuple, or a tagged expression), the following functions are provided:

`card(`d`)` The cardinality of d.

`contains?(`d, e`)` True if d contains e as an element. False otherwise.

`lower_bound(`d`)`, `upper_bound(`d`)` If d is an interval domain then they return the lower and upper bounds of d. Otherwise they are undefined.

Example Below are examples of domain expressions.

```
d1 = interval(1,2)
d2 = dom{4;6}
d3 = prod_dom(d1,d2)
d4 = sum_dom(d1,d2)
d5 = gen_prod_dom (x,y):d3 | even?(x+y) {interval(x,y)}
d6 = gen_sum_dom (x^y):d4|(x==1){interval(0,y(x))}
d7 = dom[(x,y):prod_dom(d1,d1)|(x <= y)]
```

The domain `d1` contains points indexed by the set {1,2}. The domain `d2` contains points indexed by the set {4,6}. The domain `d3` contains points indexed by the set {(1,4), (1,6), (2,4), (2,6)}. The domain `d4` contains points indexed by the set { `0^`(1,nil), `0^`(2,nil), `1^`(nil,4), `1^`(nil,6) }. The domain `d5` is equivalent to `prod_dom(interval(2,4),interval(2,6))`. The domain `d6` is equivalent to `sum_dom(interval(0,4), interval(0,6))`. The domain `d7` contains points indexed by the set {(1,1),(1,2),(2,2)}.

1.8 Data Fields

A *data field* is a function over some index domain. It assigns each point in the index domain a value. Therefore, a data field can be viewed as a set of values indexed by the underlying domain. Data field expressions have the following syntax:

dfield-exp → **df{** *expression* {(; | ;; | ;;;) *expression*} **}**
| **df** *formal* : *domain-exp* [| *filter*] **{** *expression* **}**
| **df[** *expression* | *formal* : *domain-exp* [| *filter*] **]**

Let e_0, e_1, ..., e_{n-1} be expressions, an *enumerated* one-dimensional data field is constructed by df{e_0; e_1; ...; e_{n-1}}. This data field is implicitly defined over the domain interval$(0,n-1)$. Let the identifier a be bound to such a one-dimensional data field, then a(i) denotes the value of e_i, where $0 \leq i < n$. A two-dimensional enumerated data field of size m by n is expressed by dom{$e_{0,0}$; ...; $e_{0,n-1}$;; ...;; $e_{m-1,1}$; ...; $e_{m-1,n-1}$}. This data field is implicitly defined over prod_dom(interval$(0,m-1)$,interval$(0,n-1)$). Notice that each row must be of equal length. Again, let a be bound to such a data field, then a(i,j) denotes the value of $e_{i,j}$, where $0 \leq i < m, 0 \leq j < n$. Similarly, we use ;;; for three-dimensional data fields, where each page must be of equal size. Note that there is *no* space between **df** and **{**.

Let x be a formal, d be a domain expression, e be an expression with occurrences of x, and f be a boolean expression with occurrences of x. The expression **df** x:d **{** e **}** is a data field which assigns every index point x in d with the value denoted by e. When there is a filter f, only index points passing f are assigned a value. Values on index points failing f are undefined. The expression df[e|x:d] is just another syntactic form for **df** x:d **{** e **}**. Note that there is no space between df and [.

Operators on Data Fields

Let a be an expression denoting a data field having values $a_1, a_2, \ldots, a_n$; g be an expression denoting a binary associated function; and i be the identity element of g. The following functions are provided:

domain_of(a) It denotes the domain over which a is defined.

card(a) It denotes the cardinality of domain_of(a).

compact(a) It denotes the data field defined with the same values on a new, compacted domain with undefined values and their associated index points purged.

reduce(g,i,a) It denotes the reduction of g over a. The value of the reduction is equivalent to $g(\ldots g(g(i,a_1),a_2)\ldots)$. When a is empty, the value of the reduction is defined to be i, the identity element of g as mentioned above.

scan(g,i,a) It denotes the data field of scanning g over a, where i is as defined above. If a is not one-dimensional, its canonical linear ordering is assumed for the scan.

Example Below are examples of data field expressions.

```
a1 = df{1; 2; 3}                          !!  1-d enumerated data field
a2 = df{1; 2;; 3; 4}                      !!  2-d enumerated data field
a3 = df{1; 2;; 3; 4;;; 5;6;; 7;8}         !!  3-d enumerated data field
d = prod_dom(interval(0,2),interval(0,2))
a4 = df (i,j):d { if i == 0 then a1(j) else a4(i-1,j) fi }
a5 = df[x*y|(i,j):d]
```

The data field **a1** is the set of values {1,2,3} indexed by the domain {0,1,2} and, for example, **a1(0)** is 1. The data field **a2** is the set of values {1,2,3,4} indexed by the domain {(0,0),(0,1),(1,0),(1,1)} and, for example, **a2(1,1)** is 4. The data field **a3** is the set of values {1,2,...,8} indexed by the domain {(0,0,0),(0,0,1),...,(1,1,1)} and, for example, **a3(1,0,1)** is 6. The data field **a4** is the set of values {1,2,3,1,2,3,1,2,3} indexed by the domain {(0,0),(0,1),...,(2,2)}; **a4(0,1)** is 2. The data field **a5** is the set of values {0,0,0,1,2,3,2,4,6} indexed by the domain {(0,0),(0,1),...,(2,2)} and, for example, **a5(1,2)** is 2.

1.9 Lists and Sets

Lists and sets are available in Crystal and have the following syntax:

list-exp → **list{** [*expression* {; *expression*}] **}**
set-exp → **set{** [*expression* {; *expression*}] **}**
| **set** *formal* : *domain-exp* [| *filter*] **{** *expression* **}**

Let $e_1, \ldots, e_n$ be expressions. The expression **list{** $e_1, \ldots, e_n$**}** denotes a list with enumerated elements $e_1, \ldots, e_n$. Similarly for the expression **set{** $e_1, \ldots, e_n$**}**.

Let x be a formal, d be a domain expression, e be an expression with occurrences of x, and f be a boolean expression with occurrences of x. The expression **set** x:d **{** e **}** denotes the set containing the values denoted by instantiating e with every index point x in d. When there is a filter f, only the values of those index points passing f are contained in the set.

Operators on Lists and Sets

Let l, l_1, l_2 be expressions denoting lists, e be a general expression, and f be an expression denoting a function. The following operators are defined.

null?(l) True if l is null, false otherwise.

contains?(e, l) True if l contains e as an element, false otherwise.

length(l) The number of elements in l.

head(l) The first element of l.

tail(l) The rest of l excluding the first element.

cons(e, l) The list with e being its head and l being its tail.

append(l_1, l_2) Append l_2 to l_1. This is the same as "l_1 :: l_2".

map(f, l) The list of applying f to every element of l.

Let s, s_1, s_2 be expressions denoting sets, e and f be as described above. The following operators are defined.

empty?(s) True if s is null, false otherwise.

contains?(e, l) True if s contains e as an element, false otherwise.

card(s) It denotes the number of elements in s.

union(s_1, s_2) The union of s_1 and s_2.

inter(s_1, s_2) The intersection of s_1 and s_2.

`diff`(s_1, s_2) The difference of s_1 and s_2.

`product`(s_1, s_2) The product of s_1 and s_2.

`map`(f, s) The set of applying f to every element of s.

Example Below are examples of list and set expressions.

```
l = list{1; 2; 3}
s1 = set{1; 2; 3}
s2 = set x :  interval(1,10) | even?(x) { x + 3 }
```

The list `l` and the set `s1` contain elements 1,2, and 3. The set `s2` is {5,7,9,11,13}.

1.10 Type Expressions

Type expressions provide extra information for static type checking, which catches static type errors at compile time and reduces the chance of hitting a runtime error during program execution. Though specifying the type of expressions is optional in most cases, programmers are encouraged to provide type information whenever possible. Type expressions have the following syntax:

type-exp → *constant-type* | *domain-exp* | `list_of(`*type-exp*`)` | `set_of(`*type-exp*`)`
| `sum_type(`*type-exp*`,`*type-exp*`{,`*type-exp*`})`
| `prod_type(`*type-exp*`,`*type-exp*`{,`*type-exp*`})`
| `fun_type(`*type-exp*`,`*type-exp*`)`
| `dfield_type(`*domain-exp*`,`*type-exp*`)`
constant-type → `bool` | `nat` | `int` | `float` | `string` | `char` | `domain` | `type`

The reserved words `bool`, `nat`, `int`, `float`, `string`, and `char` denote the constant types of boolean, natural numbers, integers, floating point numbers, character strings, and characters, respectively. The reserve words `domain` denotes the type of domains, and `type` denotes the universal type. Let $t, t_1, \ldots, t_n$ be expressions denoting some type, and d be a domain expression. The type expression `list-of`(t) denotes the type of all lists with elements of type t. Similarly for `set-of`(t). The type expression `prod_type`$(t_1, \ldots, t_n)$ denotes the product type of $t_1, \ldots, t_n$. Similarly for `sum_type`$(t_1, \ldots, t_n)$. The expression `dfield_type`(d, t) denotes the type of data fields defined over d with values of type t, and `fun_type`(t_1, t_2) denotes the function type from t_1 to t_2.

1.11 Basic Lexical Definitions

Here we describe the syntax of identifiers, boolean, integers, floating point numbers, and strings. We also describe the unary prefix and binary infix operators for them.

unary-op → `not` | `-`
binary-op → `+` | `-` | `*` | `/` | `**` | `mod` | `div` | `<` | `>` | `<=` | `>=` | `==` | `<>`
`and` | `or` | `::` | `@`
constant → `nil` | *boolean* | *numeric* | *character* | *string*
boolean → `true` | `false`
numeric → *integer* | *float*
integer → [`-`]*digit*{*digit*}
float → ([`-`]*digit*{*digit*}`.`{*digit*}) | ([`-`]{*digit*}`.`*digit*{*digit*})

digit → 0 | 1 | ... | 9
identifier → *alpha*{*alpha* | *digit* | *special*}
alpha → a | A | b | B | ... | z | Z
special → _ | ? | % | $ | & | |
character → # any character
string → " anything but double quote "

The unary prefix operator **not** is the boolean negation and **-** is the numeric minus sign. The binary infix operators **+**, **-**, *****, **/**, and ****** are the numeric addition, subtraction, multiplication, division, and exponentiation; **mod** and **div** are the integer modulo and integer division; **<**, **>**, **<=**, **>=**, **==**, and **<>** are the numeric comparison operators for less than, greater than, less or equal to, greater or equal to, equality testing, and inequality testing; **and** and **or** are the boolean and and or; **::** is for string and list concatenation (i.e., appending); **@** is the function composition operator. All binary infix operators are left associative and their relative precedences are defined in a way similar to the language C. Users are encouraged to use parentheses to disambiguate an expression whenever in doubt.

A string consists of all the characters appearing between 2 double quotes ("), taken as such, with the exception of the backslash character (\), which serves as an escape. The following escapes are defined: **\b** (backspace), **\f** (form feed), **\n** (newline), **\r** (carriage return), **\t** (horizontal tabulate), **\v** (vertical tabulate), **** (backslash), **\"** (double quote), **\000** to **\255** (character whose ASCII code is the decimal number appearing after the backslash (3 digits are required), **\^***c* (character corresponding to control-*c* (where *c* is any character that can be "controlled")), **\...** (all the characters (only formatting characters are allowed) between the two backslashs are ignored (as well as the backslashs off course). Examples of formatting characters are: space, newline, tab). For non alphanumeric character that don't appear in the above list, the backslash doesn't change anything. This convention for string representation is taken from Standard ML.

Characters are represented by prepending a **#** in front of the representation of the character in a string. For example, **#a** is the character *a*, **#\b** is the backspace character, **#\255** is the character whose ASCII code is 255 in decimal. Note that **##** represents the character #.

1.12 Built-in Numeric Functions

Let $x, x_1, x_2, \ldots, x_n$ be expressions denoting numeric values, and $i, i_1, i_2, \ldots, i_n$ be expressions denoting integers. The following numeric functions are built-in:

Predicates: **even?**(i), **odd?**(i) (predicates for integers testing even and odd).

Arithmetics Functions: **add**($x_1, \ldots, x_n$), **subtract**(x_1, x_2), **multiply**($x_1, \ldots, x_n$), **divide**(x_1, x_2), **negate**(x), **remainder**(i_1, i_2), **abs**(x), **gcd**(i_1, i_2), **min**($x_1, \ldots, x_n$), **max**($x_1, \ldots, x_n$), **round**(x) (round to nearest integer), **floor**(x), and **ceiling**(x).

Transcendental Functions: **exp**(x) (exponential function e^x), **log**(x) (base 2 logarithm), **ln**(x) (natural logarithm), **sin**(x), **cos**(x), **tan**(x), **asin**(x), **acos**(x), and **atan**(x).

Bitwise Logical Operators: **logand**(i_1, i_2) (bitwise logical and of i_1 and i_2), **logior**(i_1, i_2) (bitwise logical inclusive or of i_1 and i_2), **logxor**(i_1, i_2) (bitwise logical exclusive or of i_1 and i_2), **lognot**(i) (bitwise logical not of i).

2 Using the Crystal Interpreter

An interpreter based on T 3.1 has been implemented for the language defined in the previous section. It is available for anonymous ftp from /pub/Crystal/crystal-int-3.0.tar.Z on cs.yale.edu. This section describes how to use the interpreter and reminds you some potential problems in using it.

2.1 Invoking the Interpreter

Please follow the steps below to invoke the interpreter.

- Set up your shell environment variable **CRYSTAL** to the directory containing the T object codes of the Crystal interpreter. If you are at Yale CS Department and have access to the `/cs/homes` tree, then do the following:
  `setenv CRYSTAL /cs/homes/systems/ayang/crystal/obj`
- Invoke T (either from csh or from Emacs).
- Issue `(load '(crystal crint))` to load in the Crystal interpreter.
- Issue `(cr:repl)` to enter the read-eval-print-loop of the Crystal interpreter.

2.2 Interpreter Environment

The read-eval-print loop of the interpreter provides users with an interactive environment. Users can enter definitions, output expressions, or both, after the prompt and the result of the output expressions will be printed out. The following commands are also recognized by the read-eval-print-loop of the Crystal interpreter:

`load("filename")`	Load in the definitions in the Crystal program in file `filename` and evaluate all the output expressions.
`exit` and `quit`	Leave the loop and get back to T.
`help`	Print a short message about the interpreter and about how to enter multiline input.

2.3 Caveats

Robustness: This interpreter is still immature in terms of error recovery and robustness. Syntax errors are caught by the parser and the parsing will be halted immediately. The precise location of the offending token is reported. Run time errors are left to T. When hitting a run time error, the execution will break into T with little clue to help you to figure out what went wrong. Using `(backtrace)` to inspect the call stack is the most effective way for finding out what had gone wrong. Then use `(reset)` and then `(cr:repl)` to get back to the interpreter.

Size of Index Domains: Because of the way this interpreter implements index domains and data fields, their sizes should not be too big (over 10K elements). Otherwise, your program execution probably will experience many garbage collection cycles.

Case of identifiers: Because the implementation is based on T, this interpreter inherits the limitation of T that cases of identifiers are *ignored*.

Order of Definitions: Theoretically the order of the definitions in Crystal programs should not matter. However, because the way interpreter translates the definitions right now, non-function definitions should be given before they are used. This restriction is expected to go away in the near future.

3 Example Crystal Programs

If you are at Yale CS Department, there are a collection of example Crystal programs available in the directory /cs/homes/systems/ayang/crystal/examples. If you have any interesting Crystal programs that you would like to share with others, please copy them into the world-writable directory /cs/homes/systems/ayang/crystal/contrib.

These programs are included in the distribution tar file mentioned in the previous section. If you are not at Yale CS Department, please consult with the person who installed the interpreter at your site about the location of these example programs.

Below are two programs pulled from the .../**examples** directory. The first one is a simple matrix multiplication program using reduction on data fields. The second one is the bitonic sort program, also works on data fields.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!! Matrix Multiplication using reduction.                        !!!
!!! This program illustrates the use of domains, data fields, and !!!
!!      the reduction over data fields.                           !!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

n = 3
D = interval(0,n-1)
D2 = prod_dom(D,D)

A = df{ 1; 2; 3;;  4; 5; 6;;  7; 8; 9  }

B = df{ 1; 0; 0;;  0; 2; 0;;  0; 0; 3  }

C = df(i,j):D2 { reduce(add, 0, (df k:D {A(i,k)*B(k,j)})) }

!! C should be df{ 1; 4; 9;;  4;10;18;;  7;16;27  }

? C
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!! Bitonic Sort.                                                      !!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

x = df{6;2;5;9;3;1;7;4}
n = card(x)
logn = ceiling(log(n))

d = interval(0,n - 1)
e = interval(0,logn - 1)

flip = fn (x,i) : prod_type(nat,nat) { logxor(x,2**i) }
bit = fn (x,i) : prod_type(nat,nat) { logand(x,2**i) <> 0 }
xor = fn (x,y) : prod_type(nat,nat) { (x or y) and not (x and y) }

bsort = df j : interval(0,logn - 1) {
  df (i,k) : prod_dom(d,interval(0,j)) {
     if xor(bit(i,j + 1),bit(i,j - k)) then max(a,b)
        else min(a,b) fi
           where{ jprev = if k==0 then j - 1 else j fi
                  kprev = if k==0 then j - 1 else k - 1 fi
                  a = if j>0 then bsort(jprev)(i,kprev) else x(i) fi
                  b = if j>0 then bsort(jprev)(flip(i,j - k),kprev)
                             else x(flip(i,0)) fi
           }
  }
}

y = df[bsort(logn - 1)(i,logn - 1) | i : d]

? y
```

References

[1] Marina Chen, Young-il Choo, and Jingke Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 2(2):171–207, October 1988.

[2] Young-il Choo and Marina Chen. A theory of parallel-program optimization. Technical Report YALEU/DCS/TR-608, Dept. of Computer Science, Yale University, July 1988.